

# MDM: A Mode Diagram Modeling Framework for Periodic Control Systems

Zheng Wang, Geguang Pu\*, Jianwen Li, Jifeng He

Shanghai Key Laboratory of Trustworthy Computing

East China Normal University

{wangzheng, ggpu, jifeng}@sei.ecnu.edu.cn

Shenchao Qin

University of Teesside

s.qin@tees.ac.uk

Kim G. Larsen

Aalborg University of Denmark

kgl@cs.aau.dk

Jan Madsen

Technical University of Denmark

jan@imm.dtu.dk

Bin Gu

Beijing Institute of Control Engineering

gubin88@yahoo.com.cn

Periodic control systems used in spacecrafts and automobiles are usually period-driven and can be decomposed into different modes with each mode representing a system state observed from outside. Such systems may also involve intensive computing in their modes. Despite the fact that such control systems are widely used in the above-mentioned safety-critical embedded domains, there is lack of domain-specific formal modeling languages for such systems in the relevant industry. To address this problem, we propose a formal visual modeling framework called MDM as a concise and precise way to specify and analyze such systems. To capture the temporal properties of periodic control systems, we provide, along with MDM, a property specification language based on interval logic for the description of concrete temporal requirements the engineers are concerned with. The statistical model checking technique can then be used to verify the MDM models against the desired properties. To demonstrate the viability of our approach, we have applied our modeling framework to some real-life case studies from industry and helped detect two design defects for some spacecraft control system.

## 1 Introduction

The control systems that are widely used in safety-critical embedded domains, such as spacecraft control and automotive control, usually reveal periodic behaviors. Such *periodic* control systems share some interesting features and characteristics:

- They are *mode-based*. A periodic control system is usually composed of a set of modes, with each mode representing an important state of the system. Each mode either contains a set of sub-modes or performs controlled computation periodically.
- They are *computation-oriented*. In each mode, a periodic control system may perform control algorithms involving complex computations. For instance, in certain mode, a spacecraft control system may need to process intensive data in order to determine its space location.
- They behave *periodically*. A periodic control system is reactive and may run for a long time. The behavior of each mode is regulated by its own period. That is, most computations are performed within a period and may be repeated in the next period if mode switch does not take place. A mode switch may only take place at the end of a period under certain conditions.

---

\*Corresponding Author

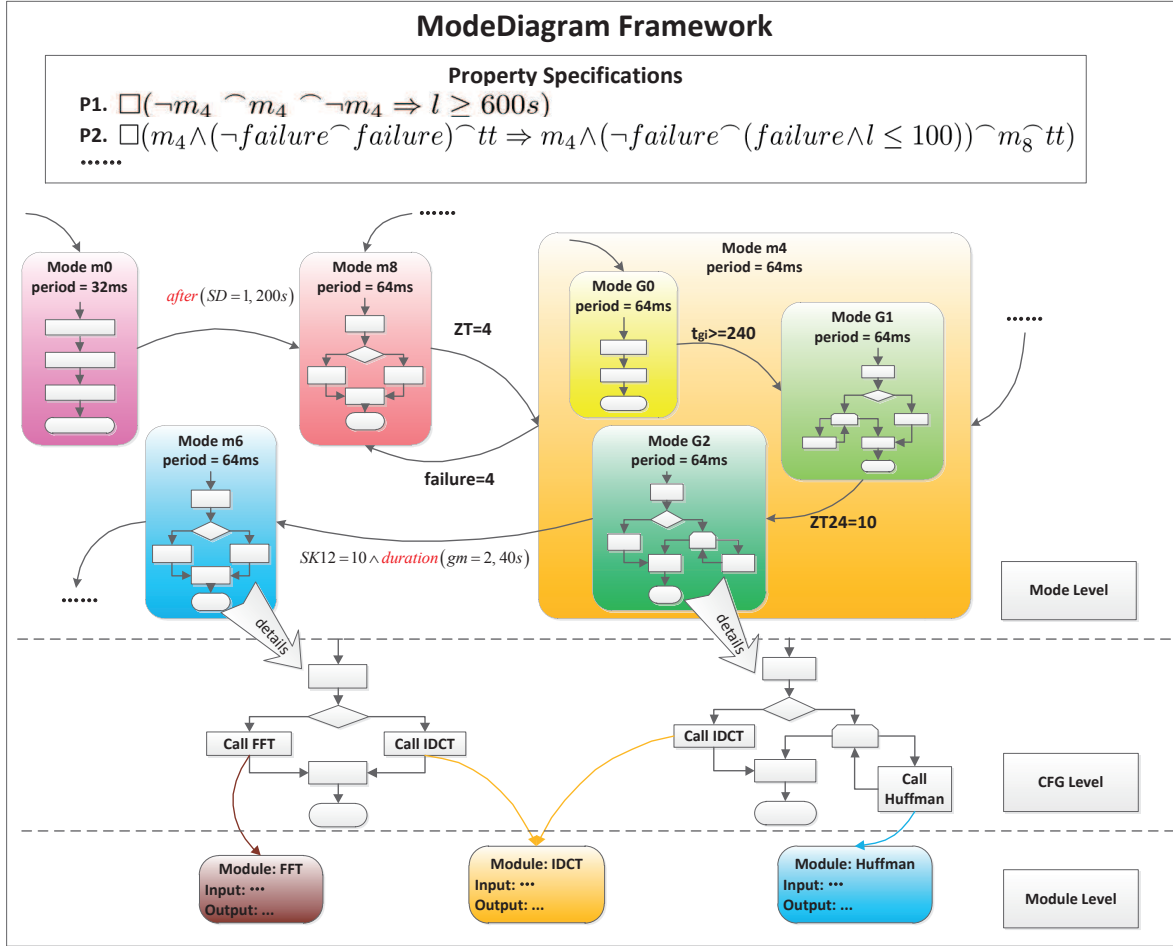


Figure 1: MDM: An (Incomplete) Example

Despite the fact that periodic control systems have been widely used in areas such as spacecraft control, there is a lack of a concise and precise domain specific formal modeling language for such systems. In our joint project with China Academy of Space Technology (CAST), we have started with several existing modeling languages but they are either too complicated therefore require too big a learning curve for domain engineers, or are too specific/general, therefore require non-trivial restrictions or extensions. This motivates us to propose a new formal but lightweight modeling language that matches exactly the need of the domain engineers, the so-called Mode Diagram Modeling framework (MDM).

Although the proposed modeling notation MDM can be regarded as a variant of Statecharts [11], it has been specifically designed to cater for the domain-specific need in modeling periodic control systems. We shall now use an example to illustrate informally the MDM framework, and leave the formal syntax and semantics to the next section. As shown in Fig 1, the key part of an MDM model is the collection of modes given in the mode level. Each mode has a period, and the periods for different modes can be different. A mode can be nested and the transitions between modes or sub-modes may take place. A transition is enabled if the associated guard is satisfied. In MDM, the transition guards may involve complex temporal expressions. For example, in the transition from mode G2 to mode m6, in addition to the condition  $SK12=10$ , it also requires that the condition  $gm=2$  has held for 40s, as captured by the duration predicate.

An MDM model is presented hierarchically. A mode that does not contain any sub-mode (termed a *leaf mode*) contains a control flow graph (CFG) encapsulating specific control algorithms or computation tasks. The details of CFGs are given in the CFG level. The CFGs may refer to modules (similar to procedures in conventional languages) details of which are given in the module level.

To support formal reasoning about MDM models, we also provide a property specification language inspired by an interval-like calculus [7], which facilitates the capture of temporal properties system engineers may be interested in. Two example properties are listed in Fig 1. The property P1 says that “whenever the system enters the m4 mode, it should stay there for at least 600s”. The formal details of the specification language is left to a later section.

To reason about whether an MDM model satisfies desired properties specified by system engineers using the property specification language, we employ statistical model checking techniques [23, 24]. Since MDM may involve complex non-linear computation in its control flow graph, complete verification is undecidable. Apart from incompleteness, statistical model checking can verify hybrid systems efficiently [4]. Our experimental results on real-life cases have demonstrated that statistical model checking can help uncover potential defects of MDM models.

In summary, we have made the following contributions in this paper:

- We propose a novel visual formal modeling notation MDM as a concise yet precise modeling language for periodic control systems. Such a notation is inspired from the industrial experiments of software engineers.
- We present a formal semantics for MDM and a property specification language to facilitate the verification process.
- We develop a new statistical model checking algorithm to verify MDM models against various temporal properties. Some real-life case studies have been carried out to demonstrate the effectiveness of the proposed framework. Furthermore, the design defects of a real spacecraft control system are discovered by our approach.

The rest of this paper is organized as follows. Section 2 presents the formal syntax and semantics of MDM. Section 3 introduces our interval-based property specification language and its semantics. The statistical model checking algorithm for MDM is developed in Section 4, followed by related work and concluding remarks.

## 2 The MDM Notations

Before developing the formal model of MDM, we will begin by giving its informal description. An MDM model is composed by several modes, variables used in the mode, and mode transitions specifying the mode switch relations. A mode essentially refers to the state of the system which can be observed from outside. The mode body can be either a Control Flow Graph (CFG), which prescribes the computational tasks the system can perform in every period, or several other modes as sub-modes. If the mode has sub-modes, when the system is in this mode, it should be in one of its sub-modes. We say that the mode is a leaf mode if its mode body is a control flow graph. A leaf mode usually encapsulates the control algorithms involving complicated computations. The CFG in a leaf mode follows the standard notation, which contains assignment, conditional and loop. It also supports function units similarly to the ones in programming languages.

$md ::= (Var^+, Mode^+, Module^+)$	$SExpr ::= Const \mid Var \mid f^{(n)}(SExpr \dots)$
$Mode ::= (name, period, initial, Body, Transition^+)$	$BTerm ::= true \mid false \mid p^{(n)}(SExpr \dots)$
$Body ::= Mode^+ \mid CFG$	$IExpr ::= (after \mid duration)(BTerm, SExpr)$
$Transition ::= (source, guard, priority, target)$	$GTerm ::= IExpr \mid BTerm$
$Module ::= (name, V_I, V_O, CFG)$	$BExpr ::= BTerm \mid \neg BExpr$
	$\mid BExpr \vee BExpr \mid BExpr \wedge BExpr$
	$guard ::= GTerm \mid \neg guard$
	$\mid guard \vee guard \mid guard \wedge guard$
(a) MDM	(b) Expressions and Guards
$CFG ::= stmts$ $stmts ::= pStmt \mid cStmt$ $pStmt ::= aStmt \mid call \ name \mid skip \mid$ $aStmt ::= x := SExpr$ $cStmt ::= stmts; stmts \mid while \ BExpr \ do \ stmts \mid$ $\quad if \ BExpr \ then \ stmts \ else \ stmts$ (c) CFG	

Figure 2: The Syntax of MDM

## 2.1 The Syntax of MDM

We briefly list its syntactical elements in Fig. 2(a). An MDM is composed of a list of modes ( $Mode^+$ ) and modules ( $Module^+$ ), as well as a list of variables ( $Var^+$ ) used in those modes and modules.

Intuitively, a mode refers to a certain state of the system which can be observed from outside. A mode has a name, a period, a body and a list of transitions. For simplicity, we assume all mode names are distinct in an MDM model. The mode period (an integer number) is used to trigger the periodic behavior of the mode. The *initial* denotes a mode is an initial mode or not. The mode body can be composed of either a control flow graph (CFG), prescribing the computational tasks the system can perform in the mode in every period, or a list of other modes as the immediate sub-modes of the current mode. If a mode has sub-modes, when the control lies in this mode, the control should also be in one of the sub-modes. A leaf mode does not have sub-modes, so its body contains a CFG. A mode is either a leaf mode, or it directly or indirectly has leaf modes as its sub-modes. A mode is called top mode if it is not a sub-mode of any other mode. The CFG in a leaf mode is the standard control flow graph, which contains nodes and structures like assignment, module call, conditional and loop. It also supports function units like the ones in conventional programming languages. The syntax of CFG is presented in Figure 2(c).

A module encapsulates computational tasks as its CFG.  $V_I$  specifies the set of variables used in the CFG, while  $V_O$  is the set of variables modified in the CFG. A module can be invoked by some modes or other modules. As a specification for embedded systems, recursive module calls are forbidden.

A transition (from  $Transition^+$ ) specifying a mode switch from one mode to another is represented as a quadruple, where the first element is the name of the source mode, the second specifies the transition condition, the third is the priority of the transition and the last element is the name of the target mode. The MDM supports mode switches at different levels in the mode-hierarchy. The transition condition (i.e. *guard*) is defined in Fig. 2(b). A state expression can be either a constant, a variable, or a real-value function on state expressions. A boolean term is either a boolean constant, or a predicate on state expressions. There are two kinds of interval expressions, after and duration. These interval expressions

are very convenient to model system behaviors related with past states. A guard term can be either an interval expression, or a boolean term. A guard is the boolean combination of guard terms. To ensure that the mode switches be deterministic, we require that the priority of a transition has to be different from the others in the same mode chain:

$$\forall m \in Mode \cdot \forall t_1, t_2 \in \text{outs}(\text{super\_modes}(md, m)) \cdot t_1 \neq t_2 \Rightarrow \text{prio}(t_1) \neq \text{prio}(t_2)$$

The functions  $\text{super\_modes}(md, m)$  and  $\text{outs}(m\text{list})$  will be defined later.

### 2.1.1 Auxiliary Definitions

Given an MDM  $md ::= (Var^+, Mode^+, Module^+)$ , we introduce two auxiliary relations:

$\text{Contains}(md) \subseteq Modes \times Modes$  for mode-subsume relation and

$\text{Trans}(md) \subseteq Modes \times Int \times guard \times Modes$  for mode-switch relation.

Given a mode  $m = (n, per, ini, b, tran)$  and a transition  $t = (m, g, pri, m')$ , we define these operations/predicates:

$$\begin{array}{lll} \text{period}(m) = per & \text{is\_initial}(m) = ini & \text{CFG}(m) = b \\ \text{prio}(t) = pri & \text{guard}(t) = g & \text{source}(t) = m \quad \text{target}(t) = m' \end{array}$$

We also define the following auxiliary functions:

$$\begin{aligned} \text{super\_modes}(md, m) &\triangleq \langle m_1, m_2, \dots, m_k \rangle, \text{ where} \\ m_k &= m \wedge m_1 \in \text{TopModes}(md) \wedge \forall 1 < i \leq k \cdot (m_{i-1}, m_i) \in \text{Contains}(md) \\ \text{and } m \in \text{TopModes}(md) &\triangleq m \in \text{Modes}(md) \wedge \neg \exists m' \cdot (m', m) \in \text{Contains}(md) \\ \text{up\_modes}(md, m, k) &\triangleq \{m_i \mid m_i \in \text{super\_modes}(md, m) \wedge \text{mod}(k, \frac{\text{period}(m_i)}{\text{period}(m)}) = 0\} \\ \text{sub\_mode}(md, m) &\triangleq m', \text{ where } (m, m') \in \text{Contains}(md) \wedge \text{is\_initial}(m') \\ \text{outs}(md, m\text{list}) &\triangleq \bigcup_{m \in m\text{list}} \{t \mid t \in \text{Trans}(md) \wedge \text{source}(t) = m\} \end{aligned}$$

The function  $\text{super\_modes}(md, m)$  retrieves a sequence of modes from a top mode to  $m$  using the  $\text{Contains}$  relation. The set  $\text{TopModes}(md)$  consists all the modes which are not sub-modes of any other mode. The function  $\text{up\_modes}(md, m, k)$  returns those modes in  $\text{super\_modes}(md, m)$  whose periods are consistent with the period count  $k$ . An MDM requires that the period of a mode should be equal to or multiple to the period of its sub-modes. The function  $\text{sub\_mode}(md, m)$  returns the initial sub-mode for a non-leaf node  $m$ , and the predicate  $\text{is\_initial}(m')$  means that the sub-mode  $m'$  is the initial sub-mode in its hierarchy. The function  $\text{outs}(m\text{list})$  returns all outgoing transitions from modes in  $m\text{list}$ .

## 2.2 The Semantics

In order to precisely analyze the behaviors of MDM, for instance, model checking of MDM, we need its formal semantics. In this section, we present the operational semantics for MDM.

$\sigma_1 \dots \sigma_n \models b$	$\Leftrightarrow \sigma_n \models b$
$\sigma_1 \dots \sigma_n \models \neg g$	$\Leftrightarrow \neg(\sigma_1 \dots \sigma_n \models g)$
$\sigma_1 \dots \sigma_n \models g_1 \vee g_2$	$\Leftrightarrow \sigma_1 \dots \sigma_n \models g_1 \text{ or } \sigma_1 \dots \sigma_n \models g_2$
$\sigma_1 \dots \sigma_n \models g_1 \wedge g_2$	$\Leftrightarrow \sigma_1 \dots \sigma_n \models g_1 \text{ and } \sigma_1 \dots \sigma_n \models g_2$
$\sigma_1 \dots \sigma_n \models \text{duration}(b, l)$	$\Leftrightarrow \sigma_n(l) = v \wedge \exists i < n \cdot (\sigma_i(ts) + v \leq \sigma_n(ts) \wedge \sigma_{i+1}(ts) + v \geq \sigma_n(ts) \wedge \forall i \leq j \leq n \cdot \sigma_j(b) = \text{true})$
$\sigma_1 \dots \sigma_n \models \text{after}(b, l)$	$\Leftrightarrow \sigma_n(l) = v \wedge \exists i < n \cdot (\sigma_i(ts) + v \leq \sigma_n(ts) \wedge \sigma_{i+1}(ts) + v \geq \sigma_n(ts) \wedge \sigma_i(b) = \text{true})$

Table 1: The Interpretation of Guards

### 2.2.1 Configuration

The configuration in our operational semantics is represented as  $(md, m, l, pc, k, \Sigma)$ , where

- $md$  is the MDM, and  $m$  is the mode the system control currently lies in.
- $l \in \{\text{Begin}, \text{Execute}, \text{End}\}$  specifies the system is in the beginning, middle, or end of a period.
- $pc \in \mathcal{L}$ , where  $\mathcal{L} = \mathcal{N} \cup \{\text{Start}, \text{Exit}, \perp\}$  is the program counter to execute the control flow graph.  $\mathcal{N}$  is used to represent the nodes in control flow graphs and *Start*, *Exit* denote the start and exit locations of a control flow graph respectively. If the current mode is not equipped with any flow graph, we use the symbol  $\perp$  as a placeholder.
- The fourth component  $k$  records the count of periods for the current mode. If the system switches to another mode, it will be reset to 1. The period count is used to distinguish whether a super-mode of the current mode is allowed to check its mode switch guard.
- $\Sigma$  is a list of states of the form  $\Sigma' \cdot \sigma$ , where  $\sigma$  denotes the current state ( $\sigma \in \text{State} \triangleq \text{Vars} \rightarrow \mathbb{R}$ ) and  $\Sigma'$  represents a history of states.

**Guards** The evaluation of a transition guard may depend on the current state as well as some historical states. Table 1 shows how to interpret a guard in a given sequence of states. The symbol *ts* is the abbreviation of the variable *timestamp*. The guard  $\text{duration}(b, l)$  evaluates to true if the boolean expression  $b$  has been true during the time interval  $l$  up to the current moment. The guard  $\text{after}(b, l)$  evaluates to true if the boolean expression  $b$  was true the time interval  $l$  ago. In this table,  $b$  is a pure boolean expression without interval expressions and  $l$  is a state expression.

### 2.2.2 Operational Rules

The operational rules for MDM are given in Table 2. Here we adopt a big-step operational semantics for MDM, which means that we only observe the start and end points of a period in the current mode, while the state changes within a period are not recorded. This is reasonable since in practice engineers usually monitor the states at the two ends of a period to decide if it works well. In the rules, we make use of an auxiliary function *execute* to represent the execution results for the mode in one period.

$$\text{execute} : \mathcal{CFG}(V) \times \mathcal{L} \times \text{State} \times \mathbb{R}^+ \rightarrow \mathcal{L} \times \text{State}$$

It receives a flow graph, a program counter, an initial state and the time permitted to execute and returns the state and program counter after the given time is expired. Its detailed definition is left in the report [22]. We now explain the operational rules:

1. (ENTER). When the system is at the beginning of a period, if the current mode  $m$  has sub-modes, the system enters the initial sub-mode of  $m$ .

(ENTER)	$\frac{\text{CFG}(m) = \perp}{(md, m, \text{Begin}, \perp, k, \Sigma) \longrightarrow (md, m', \text{Begin}, pc', k, \Sigma)}$ <p style="text-align: center;">where <math>m' = \text{sub\_mode}(md, m)</math> and <math>pc' = \begin{cases} \perp, &amp; \text{if } \text{CFG}(m') = \perp \\ \text{Start}, &amp; \text{if } \text{CFG}(m') \neq \perp \end{cases}</math></p>
(DETECT)	$\frac{\text{CFG}(m) \neq \perp}{(md, m, \text{Begin}, pc, k, \Sigma \cdot \sigma) \longrightarrow (md, m, \text{Execute}, pc, k, \Sigma \cdot \text{sampling}(\sigma))}$
(EXECUTE)	$\frac{\text{execute}(\text{CFG}(m), pc, \sigma, \text{period}(m)) = (pc', \sigma')}{(md, m, \text{Execute}, pc, k, \Sigma \cdot \sigma) \longrightarrow (md, m, \text{End}, pc', k, \Sigma')}$ <p style="text-align: center;">where <math>\Sigma' = \Sigma \cdot \sigma'[ts \mapsto \sigma(ts) + \text{period}(m)]</math>  <math>pc \neq \text{Exit}</math></p>
(CONTINUE)	$\frac{pc \neq \text{Exit}}{(md, m, \text{End}, pc, k, \Sigma) \longrightarrow (md, m, \text{Execute}, pc, k, \Sigma)}$
(REPEAT)	$\frac{\forall t \in \text{outs}(\text{up\_modes}(md, m, k)) \cdot \Sigma \not\models \text{guard}(t)}{(md, m, \text{End}, \text{Exit}, k, \Sigma) \longrightarrow (md, m, \text{Begin}, \text{Start}, k+1, \Sigma)}$
(SWITCH)	$\frac{\begin{array}{l} \exists t \in \text{outs}(md, \text{up\_modes}(md, m, k)) \cdot \Sigma \models \text{guard}(t) \wedge \\ \forall t' \in \text{outs}(\text{up\_modes}(md, m, k)) - \{t\} \cdot (\Sigma \not\models \text{guard}(t') \vee \text{prio}(t') < \text{prio}(t)) \end{array}}{(md, m, \text{End}, \text{Exit}, k, \Sigma) \longrightarrow (md, m', \text{Begin}, pc', 1, \Sigma)}$ <p style="text-align: center;">where <math>m' = \text{target}(t)</math> and <math>pc' = \begin{cases} \perp, &amp; \text{if } \text{CFG}(m') = \perp \\ \text{Start}, &amp; \text{if } \text{CFG}(m') \neq \perp \end{cases}</math></p>

Table 2: Operational Semantic Rules for MDM

2. (DETECT). When the system is at the beginning of a period, if the current mode  $m$  is a leaf mode, the system updates its state by sampling from sensors. The function *sampling* represents the side-effect on variables during sensor detection. The period label  $l$  is changed to be *Execute*, indicating that the system will then perform computational tasks specified by the control flow graph of  $m$ .
3. (EXECUTE). This rule describes the behaviors of executing CFG of the leaf mode  $m$ . The function *execute* is used to compute the new state  $\sigma'$  from  $\sigma$ . The computation task may be finished in the current period and  $pc' = \text{Exit}$  holds or the task is not finished and the program counter points to some location in the control flow graph. The value of the timestamp variable  $ts$  in  $\sigma'$  is equal to its value in state  $\sigma$  plus the period of the mode  $m$ .
4. (CONTINUE). This rule tells that when the computation task in leaf mode is not finished in a period, it will continue its task in the next period. In this case, the system is implicitly not allowed to switch to other modes from the current mode. When moving to the next period, sensor detection is skipped.
5. (REPEAT). This rule specifies the behavior of restarting the flow graph when the computation task is finished in a period. When it is at the end of a period and the system finishes executing the flow graph ( $pc = \text{Exit}$ ), if there is no transition guard enabled, the system stays in the same mode and restarts the computation specified by the flow graph.
6. (SWITCH). This rule specifies the behavior of the mode transition. There exists a transition  $t$ ,

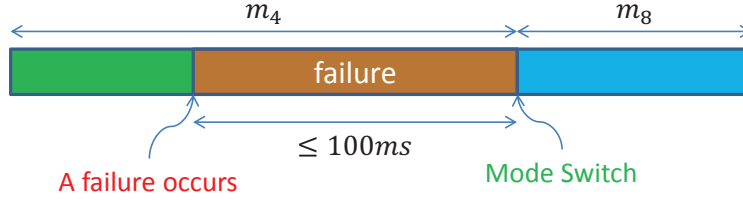


Figure 3: A Property about Failure

$$\begin{array}{ll}
 \text{Terms} & \theta \triangleq r \mid v \mid l \mid f(\theta_1, \dots, \theta_n) \\
 \text{Formulas} & \phi, \psi \triangleq tt \mid ff \mid p(\theta_1, \dots, \theta_n) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \frown \psi
 \end{array}$$

Figure 4: The Syntax of ITL

whose guard holds on the sequence of states  $\Sigma$ . And the priority of  $t$  is higher than that of any other enabled transitions.

### 3 The Property Specification Language

We adopt the Interval Temporal Logic (ITL) [17] as the property specification language. The reason why we adopt the interval-based logic instead of state-based logics like LTL or CTL is that most of the properties the domain engineers care about are related to some duration of time. For instance, the engineers would like to check if the system specified by MDM can stay in a specific state for a continuous period of time instead of just reaching this state. Another typical scenario illustrated in Fig. 3 is that, “when the system control is in mode  $m_4$ , if a failure occurs, it should switch to mode  $m_8$  in 100 ms”. The standard LTL formula  $\Box(failure \wedge m_4 \Rightarrow \Diamond m_8)$  can be used to specify that “when the system is in mode  $m_4$ , and a failure occurs, it should switch to mode  $m_8$ ”. But the real-time feature “in 100 ms” is lost. Though the extensions of LTL or CTL may also describe the interval properties to some extent, it is more natural for the domain engineers to use interval-based logic since the intuitive chop operator ( $\frown$ ) is available in ITL.

An interval logic formula can be interpreted over a time interval [7] or over a “state interval” (a sequence of states) [17]. As explained later in this section, our proposed specification language will be interpreted in the latter way [17] except for a small modification on the interpretation of the chop operator ( $\frown$ ).

#### 3.1 Syntax

The syntax of the specification language is defined in Fig 4, where

- The set of terms  $\theta$  contains real-value constants  $r$ , temporal variables  $v$ , a special variable  $l$ , and functions  $f(\theta_1, \dots, \theta_n)$  (with  $f$  being an  $n$ -arity function symbol and  $\theta_1, \dots, \theta_n$  being terms).
- Formulae can be boolean constants ( $tt, ff$ ), predicates ( $p(\theta_1, \dots, \theta_n)$  with  $p$ , an  $n$ -arity predicate symbol), classical logic formulae (constructed using  $\neg, \wedge$ , etc), or interval logic formulae (constructed using  $\frown$ ). If the formula  $\phi \frown \psi$  holds for an interval  $\ell$ , it means that the interval  $\ell$  can be “chopped” into two sub-intervals, where  $\phi$  holds for the first sub-interval and  $\psi$  holds for the second one.



$\mathcal{I}_{\mathcal{T}}(r, \Sigma) = r$	
$\mathcal{I}_{\mathcal{T}}(l, \Sigma) = \begin{cases} \sigma_{n-1}(ts) - \sigma_0(ts) & \text{if } \Sigma = \sigma_0 \dots \sigma_{n-1} \\ \infty & \text{if }  \Sigma  = \infty \end{cases}$	
$\mathcal{I}_{\mathcal{T}}(v, \sigma_0, \Sigma) = \sigma_0(v)$	
$\mathcal{I}_{\mathcal{T}}(f(\theta_1, \dots, \theta_n), \Sigma) = f(\mathcal{I}_{\mathcal{T}}(\theta_1, \Sigma), \dots, \mathcal{I}_{\mathcal{T}}(\theta_n, \Sigma))$	
$\mathcal{I}_{\mathcal{F}}(p(\theta_1, \dots, \theta_n), \Sigma) = \text{true}$	iff $p(\mathcal{I}_{\mathcal{T}}(\theta_1, \Sigma), \dots, \mathcal{I}_{\mathcal{T}}(\theta_n, \Sigma))$
$\mathcal{I}_{\mathcal{F}}(tt, \Sigma) = \text{true}$	iff <i>always</i>
$\mathcal{I}_{\mathcal{F}}(ff, \Sigma) = \text{false}$	iff <i>always</i>
$\mathcal{I}_{\mathcal{F}}(\neg\phi, \Sigma) = \text{true}$	iff $\mathcal{I}_{\mathcal{F}}(\phi, \Sigma) = \text{false}$
$\mathcal{I}_{\mathcal{F}}(\phi \wedge \psi, \Sigma) = \text{true}$	iff $\mathcal{I}_{\mathcal{F}}(\phi, \Sigma) = \text{true}$ and $\mathcal{I}_{\mathcal{F}}(\psi, \Sigma) = \text{true}$
$\mathcal{I}_{\mathcal{F}}(\phi \frown \psi, \Sigma) = \text{true}$	iff $\exists k < \infty. \Sigma = (\sigma_0 \dots \sigma_k \cdot \Sigma') \wedge \mathcal{I}_{\mathcal{F}}(\phi, \sigma_0 \dots \sigma_k) = \text{true} \wedge \mathcal{I}_{\mathcal{F}}(\psi, \Sigma') = \text{true}$

Table 3: Interpretation of the Specification Language

As a kind of temporal logic, ITL also provides the  $\Box$  and  $\Diamond$  operators. They are defined as the abbreviations of  $\frown$ .

$$\Diamond\phi \triangleq tt \frown (\phi \frown tt), \text{ for some sub-interval } , \quad \Box\phi \triangleq \neg\Diamond(\neg\phi), \text{ for all sub-intervals}$$

By the specification language proposed here, we can describe the properties the domain engineers may desire. For instance, the following property describes the scenario shown in Fig. 3.

$$\Box(m_4 \wedge (\neg failure \frown failure) \frown tt \Rightarrow m_4 \wedge (\neg failure \frown (failure \wedge l \leq 100)) \frown m_8 \frown tt)$$

### 3.2 Interpretation

Terms/formulae in our property specification language are interpreted in the same way as in Maszkowski [17], where an interval is represented by a finite or infinite sequence of states ( $\Sigma = \sigma_0 \sigma_1 \dots \sigma_{n-1} \dots$ ), where  $\sigma_i \in \text{State}$ . The interpretation is given by two functions (1) term interpretation:  $\mathcal{I}_{\mathcal{T}} \in \text{Terms} \times \text{Intv} \mapsto \mathbb{R}$ , and (2) formula interpretation function:  $\mathcal{I}_{\mathcal{F}} \in \text{Formulas} \times \text{Intv} \mapsto \{\text{true}, \text{false}\}$ . Table 3 defines these two functions, where  $ts$  denotes the variable *timestamp*. The value of the variable *timestamp* increases with the elapse of the time. i.e., for any two states in the same interval  $\sigma_i, \sigma_j$ , if  $i < j$ , then  $\sigma_i(ts) < \sigma_j(ts)$ . Thus, we can compute the length of time interval based on the difference of the two time stamps located in the first and last states respectively. The interpretation of a variable  $v$  on  $\Sigma$  is the evaluation of  $v$  on the first state of  $\Sigma$ . Note that our chop operator requires that the first sub-interval of  $\Sigma$  is restricted to be finite no matter whether the interval  $\Sigma$  itself is finite or not.

## 4 MDM Verification by Statistical Model Checking

As a modelling & verification framework for periodic control systems, MDM supports the modeling of periodic behaviors, mode transition, and complex computations involving linear or non-linear mathematical formulae. Moreover, it also provides a property specification language to help the engineers capture requirements. In this section, we will show how to verify that an MDM model satisfies properties formalized in the specification language. There are two main obstacles to apply classic model checking

techniques on MDM: (1) MDM models involve complex computations like non-linear mathematic formulae; (2) MDM models are open systems which need intensive interactions with the outside.

Our proposed approach relies on Statistical Model Checking (SMC) [20, 23, 16, 6]. SMC is a simulation-based technique that runs the system to generate traces, and then uses statistical theory to analyze the traces to obtain the verification estimation of the entire system. SMC usually deals with the following quantitative aspect of the system under verification [23]:

What is the probability that a random run of a system will satisfy the given property  $\phi$ ?

Since the SMC technique verifies the target system with the probability estimation instead of the accurate analysis, it is very effective when being applied to open and non-linear systems. Because SMC depends on the generated traces of the system under verification, we shall briefly describe how to simulate an MDM and then present an SMC algorithm for MDM.

## 4.1 MDM Simulation

The MDM model captures a reactive system [10]. The MDM model executes and interacts with its external environment in a *control loop* in one period as follows: (1) Accept inputs via sensors from the environment. (2) Perform computational tasks. (3) Generate outputs to drive other components. The MDM simulation engine simulates the process of the control loop above.

Generally speaking, the simulation is implemented according to the inference rules defined in Table 2. However, the behaviors of an MDM model depends not only on the MDM itself, but also on the initial state and the external environment. When we simulate the MDM model, the initial values are randomly selected from a range specified by the control engineers from CAST. As a specification language, the type of variables defined in MDM can be real number. To implement the simulation, we use float variables instead, which may introduce some problems on precision. There are lots of techniques can be adopted to check if any loss of precision may cause problems[14]. Because the simulation doesn't take care of the platform to deploy the system specified by the MDM, the time during simulation is not the real time, but the logic time. For each iteration in the *control loop*, the time is increased by the length of period of the current mode.

To make the simulation be executable, we have to simulate the behaviors of the environment to make the MDM model to be closed with its environment. The environment simulator involving kinematic computations designed by the control engineers is combined with the MDM to simulate the physical environment the MDM model interacts with. In the beginning of each period, the simulator checks whether there are sub-modes in the current mode. If so, the simulator takes the initial sub-mode as the new current mode. When the current mode is a leaf mode, the simulator invokes the library simulating the physical environments and updates the internal state by getting the value detected from sensors. Then the simulator executes the control flow graph in the leaf mode. We assume that there is enough time to execute the CFG. The situation that tasks are allowed not to be finished in one period is not considered during simulation. In the end of each period, the guards of transitions are checked. The satisfactions of duration and after guards do not only depend on the current state, but also the past states. The simulator sets a counter for each duration/after guard instead of recording the past states. As an MDM model is usually a non-terminating periodic system, the bound of periods is set during the process of simulation.

## 4.2 SMC Algorithm

We apply the methodology in [23] to estimate the probability that a random run of an MDM will satisfy the given property  $\phi$  with a certain precision and certain level of confidence. The statistical model check-

```

input   md: the MDM,  $\phi$ : property, B: bound of periods
          $\delta$ : confidence,  $\varepsilon$ : approximation
output  p: the probability that  $\phi$  holds on an arbitrary run of md
begin
10   $N := 4 * \frac{\log \frac{1}{\delta}}{\varepsilon^2}, a := 0$ 
20  for  $i := 1$  to  $N$  do
30    generate an initial state  $s_0$  randomly
40    simulate the MDM from  $s_0$  in  $B$  periods to get the state trace  $\Sigma$ 
50    if  $(\mathcal{I}_{\mathcal{T}}(\phi, \Sigma) = \text{true})$  then  $a := a + 1$ 
60  end for
70  return  $\frac{a}{N}$ 
end

```

Figure 5: Probability Estimation for MDM

ing algorithm for MDM is illustrated in Fig. 5. Since the run of the MDM usually is infinite, the users can set the length of the sequence by the number of periods based on the concrete application. This algorithm firstly computes the number  $N$  of runs based on the formula  $N := 4 * \log(1/\delta)/\varepsilon^2$  which involves the confidence interval  $[p - \delta, p + \delta]$  with the confidence level  $1 - \varepsilon$ . Then the algorithm generates the initial state (line 30) and gets a state trace  $\Sigma$  by the inference rules defined in Table 2 (line 40). The algorithm in line 50 decides whether  $\phi$  holds on the constructed interval based on the interpretation for the specification language mentioned in Section 3. If the interpretation is true, the algorithm increases the number of traces on which property  $\phi$  holds. Line 70 returns the probability for the satisfaction of  $\phi$  on the MDM.

### 4.3 Experiments

We have implemented the MDM modeling and verification framework and applied it onto several real periodic control systems. The implementation framework of SMC is illustrated in Fig. 6, where the simulator is used to simulate the MDM by the proposed operational semantics and the generated traces are for the statistical model checker. One of the real periodic control systems (termed as *A*) is for spacecraft control developed by Chinese Academy of Space Technology. Fig. 1(shown in Section 1) is a small portion of the MDM model for system *A*.

We communicate with the engineers in CAST, summarize several properties the two models of spacecrafts should obey, and present these properties in our specification language. A total of 12 properties are developed by the engineers and these properties are verified on the systems *A*. We only highlight three properties because the verification results on these three properties reveal two defects.

- *After 3000 seconds, the system will eventually reach the stable state forever*

$$\ell \geq 3000 \Rightarrow tt \frown \Box(\sqrt{\omega_x^2 + \omega_y^2 + \omega_z^2} \leq 0.1 \wedge \sqrt{\dot{\omega}_x^2 + \dot{\omega}_y^2 + \dot{\omega}_z^2} \leq 0.01)$$

where  $\omega_x$ ,  $\omega_y$  and  $\omega_z$  are angles.  $\dot{\omega}_x$ ,  $\dot{\omega}_y$  and  $\dot{\omega}_z$  are angle rates.

- *The system starts from mode  $m_0$ , and then it will finally switch to mode  $m_5$  or  $m_6$  or  $m_8$ , and stay in one of these three modes forever*

$$(\text{mode} = 0) \frown tt \frown \Box(\text{mode} = 5 \vee \text{mode} = 6 \vee \text{mode} = 8)$$

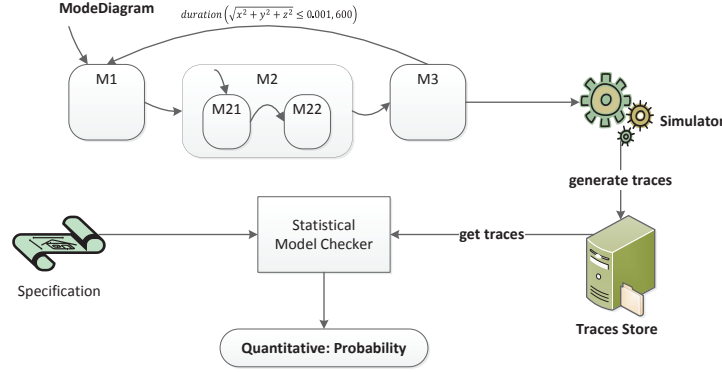


Figure 6: The Framework of Implementation

- Whenever the system switches to mode  $m4$  and then leaves  $m4$ , during its stay in  $m4$ , it firstly stays in sub-mode  $G0$ , and then it switches to sub-mode  $G1$ , and then  $G2$ .

$$\Box(\text{mode} \neq 4 \wedge \text{mode} = 4 \wedge \text{mode} \neq 4 \Rightarrow \text{mode} \neq 4 \wedge \text{mode} = 4 \wedge (\text{gm} = 0 \wedge \text{gm} = 1 \wedge \text{gm} = 2) \wedge tt)$$

For the parameters of the statistical model checking algorithm, we set the half length of confident interval to be 1% ( $\delta = 1\%$ ) and the error rate to be 5% ( $\varepsilon = 5\%$ ). Based on this algorithm, the total 7369 traces for each control system are required to be generated to compute the probabilities during the verification process.

During the verification phase by the statistical model checking on MDM, two design defects in system A are uncovered by analyzing the verification results: (1) A variable is not initialized properly. (2) A value from sensors is detected from the wrong hardware address. In the traditional developing process in CAST, these two defects may be revealed only after a prototype of the software is developed and then tested. Our approach can find such bugs in design phase and reduce the cost to fix defects.

## 5 Related Work

Our MDM can be broadly considered as a variant of Statecharts [11], where a mode in MDM is similar to a state in the Statecharts. However, we note the following distinctions: (1) In Statecharts, when a transition guard holds, the system immediately switches to the target state. But in MDM, mode switches are only allowed to be triggered at the end of a period. (2) In Statecharts, a transition guard is usually a boolean expression on the current(source) state; while in MDM, transition guards may involve past states via predicates like during and after. (3) In Statecharts, all observations on the system are the states; while MDM also concerns about the computation aspect of the system by means of the flow graphs provided in the leaf modes.

Timed Automata are a modeling tool for the description and verification of real-time systems [3, 5]. It provides the *clock* variable to support the time explicitly. Timed Automata only focus on the linear computation for time since it has nice time zone semantics supporting the timed verification. Hybrid Automata [2] extend the traditional automata to deal with complex computation like the difference and differentiation while it is not a systematic modeling tool which supports the rich modeling mechanisms like the hierarchy, types etc.

Giese et al. [8] have proposed a semantics of real-time variant of Statecharts by introducing the Hierarchical Timed Automata. In another work [9] they have presented a compositional verification approach to the real time UML designs. A. K. Mok et al. have developed a kind of hierarchical real-time chart named “Modechart” [15]. Compared with Giese et al. [8], parallel modes are supported in Modechart.

Stateflow is the Statechart-like language used in the commercial software Matlab/Simulink [1]. The Stateflow language enriches Statecharts to allow it to support flow-based and state-based computations together for specifying discrete event systems. Our MDM focuses more on periodic control systems, which can be regarded as a specific type of discrete event systems, and it provides the first class element *period* to facilitate the precise modeling of periodic-driven systems. The transitions in Stateflow can be attached with a flowchart to describe complicated computation, the MDM specifies the flow graph for the computation in its leaf modes. While Stateflow focuses only on the modeling aspect of the systems, the MDM integrates modeling and reasoning by providing a property specification language with a verification algorithm.

Some researchers introduce *operational modes* [18, 19] during the modeling in hardware/software co-synthesis. The operational mode is essentially a state in the automaton, but it can be attached a flowchart for the description of the computation. It does not support the nested mode and period explicitly. However, it is actually an informal modeling notation because it allows to specify the system behaviors in natural language. Our MDM is a lightweight formal notation for the modeling with its precise operational semantics.

Giotto is also a periodic-driven modeling language proposed by Henzinger et al. [13]. The main difference between Giotto and MDM is the computation mechanism provided. The tasks in a mode can be performed in parallel in Giotto while the details of the tasks are omitted and are moved to the implementation stage. The MDM supports the detailed description of the computation in their leaf modes since the design of it is targeted for control systems which may involve rich algorithms. The MDM does not support the parallel computation explicitly at present since it could bring the nondeterminism at the design level. The emphasis of the Giotto is more for the modeling and synthesis of parallel tasks while the MDM is for the modeling and verification based on the proposed specification language.

Runtime Verification is a verification approach based on extracting information by executing the system and using the information to detect whether the observed behaviors violating the expected properties [12, 21]. The verification approach we apply in this paper is also a kind of runtime verification. But our methodology is the off-line analysis, while [21] applies an on-line monitoring approach using Aspect-J. The reason to propose off-line analysis is that the cost to decide if an ITL formula is satisfiable on a given trace is expensive, so information extraction and analysis are separated to two phases in our approach.

## 6 Conclusion

In this paper, we propose the Mode Diagram Modeling framework (MDM), a domain-specific formal visual modeling language for periodic control systems. To support formal reasoning, MDM is equipped with a property specification language based on interval temporal logic and a statistical model checking algorithm. The property specification language allows engineers to precisely capture various properties they desire, while the verification algorithm allows them to reason about MDM models with respect to those properties. The viability and effectiveness of the proposed MDM framework have been demonstrated by a number of real-life case studies, where defects of spacecraft control systems have been

uncovered in the early design stage.

## Acknowledgement

WANG Zheng and GU Bin are partially supported by Projects NSFC No. 90818024 and NSFC No. 91118007. PU Geguang is partially supported by NSFC Projects No. 61061130541 and No. 61021004. Jianwen Li is partially supported by Shanghai Knowledge Service Platform for Trustworthy Internet of Things (No. ZF1213). Jifeng He is partially supported by 973 Project No. 2011CB302904. Shengchao Qin was supported in part by EPSRC project EP/G042322.

## References

- [1] *The Mathworks: Stateflow and Stateflow Coder, User's Guide*. Available at [www.mathworks.com/help/releases/R13sp2/pdf\\_doc/stateflow/sf Ug.pdf](http://www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf Ug.pdf).
- [2] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger & Pei-Hsin Ho (1992): *Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems*. In Robert L. Grossman, Anil Nerode, Anders P. Ravn & Hans Rischel, editors: *Hybrid Systems, Lecture Notes in Computer Science 736*, Springer, pp. 209–229, doi:10.1007/3-540-57318-6\_30.
- [3] Rajeev Alur & David L. Dill (1994): *A theory of timed automata*. *Theor. Comput. Sci.* 126, pp. 183–235, doi:10.1016/0304-3975(94)90010-8. Available at <http://dl.acm.org/citation.cfm?id=180782.180519>.
- [4] Ananda Basu, Saddek Bensalem, Marius Bozga, Benoît Delahaye & Axel Legay (2012): *Statistical abstraction and model-checking of large heterogeneous systems*. *STTT* 14(1), pp. 53–72, doi:10.1007/s10009-011-0201-2.
- [5] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson & Wang Yi (2011): *Developing UPPAAL over 15 years*. *Softw., Pract. Exper.* 41(2), pp. 133–142, doi:10.1002/spe.1006.
- [6] Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis & Zheng Wang (2011): *Time for Statistical Model Checking of Real-Time Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV, Lecture Notes in Computer Science 6806*, Springer, pp. 349–355, doi:10.1007/978-3-642-22110-1\_27.
- [7] Bruno Dutertre (1995): *Complete Proof Systems for First Order Interval Temporal Logic*. In: *LICS*, IEEE Computer Society, pp. 36–43, doi:10.1093/logcom/14.2.215.
- [8] Holger Giese & Sven Burmester (2003): *Real-Time Statechart Semantics*. Technical Report TR-RI-03-239, Software Engineering Group, University of Paderborn, Warburger Str. 100, D-33098 Paderborn, Germany. Available at <http://www.hpi.uni-potsdam.de/giese/gforge/publications/tr-ri-03-239.pdf>.
- [9] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer & Stephan Flake (2003): *Towards the compositional verification of real-time UML designs*. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, ESEC/FSE-11*, ACM, New York, NY, USA, pp. 38–47, doi:10.1145/940071.940078.
- [10] D. Harel & A. Pnueli (1985): *On the development of reactive systems*, pp. 477–498. Springer-Verlag New York, Inc., New York, NY, USA. Available at <http://dl.acm.org/citation.cfm?id=101969.101990>.
- [11] David Harel (1987): *Statecharts: A visual formalism for complex systems*. *Science of Computer Programming* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.
- [12] Klaus Havelund (2008): *Runtime Verification of C Programs*. In: *Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems: 8th International Workshop, TestCom '08 / FATES '08*, Springer-Verlag, Berlin, Heidelberg, pp. 7–22, doi:10.1007/978-3-540-68524-1\_3.

- [13] Thomas A. Henzinger, Benjamin Horowitz & Christoph M. Kirsch (2001): *Giotto: a Time-triggered Language for Embedded Programming*. Technical Report, Department of Electronic Engineering and Computer Science, University of California, Berkeley, Berkeley, CA, USA. Available at <http://mtc.epfl.ch/~tah/Publications/giotto.pdf>.
- [14] Nicholas J. Higham (2002): *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM, doi:10.1137/1.9780898718027.
- [15] Farnam Jahanian & Aloysius K. Mok (1994): *Modechart: A Specification Language for Real-Time Systems*. *IEEE Trans. Softw. Eng.* 20, pp. 933–947, doi:10.1109/32.368134. Available at 10.1109/32.368134.
- [16] Kim G. Larsen, Axel Legay, Louis-Marie Traonouez & Andrzej Wasowski (2011): *Robust Specification of Real Time Components*. In Uli Fahrenberg & Stavros Tripakis, editors: *FORMATS, Lecture Notes in Computer Science* 6919, Springer, pp. 129–144, doi:10.1007/978-3-642-24310-3\_10.
- [17] Ben C. Moszkowski & Zohar Manna (1983): *Reasoning in Interval Temporal Logic*. In Edmund M. Clarke & Dexter Kozen, editors: *Logic of Programs, Lecture Notes in Computer Science* 164, Springer, pp. 371–382, doi:10.1007/3-540-12896-4\_374.
- [18] Hyunok Oh & Soonhoi Ha (2002): *Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints*. In: *Proceedings of the tenth international symposium on Hardware/software codesign, CODES '02*, ACM, New York, NY, USA, pp. 133–138, doi:10.1145/774789.774817.
- [19] Marcus T. Schmitz, Bashir M. Al-Hashimi & Petru Eles (2005): *Cosynthesis of energy-efficient multimode embedded systems with consideration of mode-execution probabilities*. *IEEE Trans. on CAD of Integrated Circuits and Systems* 24(2), pp. 153–169, doi:10.1109/TCAD.2004.837729.
- [20] Koushik Sen, Mahesh Viswanathan & Gul Agha (2004): *Statistical Model Checking of Black-Box Probabilistic Systems*. In Rajeev Alur & Doron Peled, editors: *CAV, Lecture Notes in Computer Science* 3114, Springer, pp. 202–215, doi:10.1007/978-3-540-27813-9\_16.
- [21] Volker Stolz & Eric Bodden (2006): *Temporal Assertions using AspectJ*. *Electron. Notes Theor. Comput. Sci.* 144, pp. 109–124, doi:10.1016/j.entcs.2006.02.007.
- [22] Zheng Wang, Geguang Pu, Shengchao Qin, Jianwen Li, Kim G. Larsen, Jan Madsen, Bin Gu & Jifeng He (2011): *ModeDiagram: A Modeling Notation for Requirement Analysis in Aerospace*. Technical Report LAB-205-TR-HT-11-0812, Software Engineering Insititute, East China Normal University, North Zhongshan Road. 3663, Shanghai, China. Available at <http://www.lab205.org/MDM/reports.html>.
- [23] Håkan L. S. Younes (2005): *Probabilistic Verification for “Black-Box” Systems*. In Kousha Etessami & Sriram K. Rajamani, editors: *CAV, Lecture Notes in Computer Science* 3576, Springer, pp. 253–265, doi:10.1007/11513988\_25.
- [24] Håkan L. S. Younes & Reid G. Simmons (2002): *Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling*. In Ed Brinksma & Kim Guldstrand Larsen, editors: *CAV, Lecture Notes in Computer Science* 2404, Springer, pp. 223–235, doi:10.1007/3-540-45657-0\_17.